02 | Mutex：庖丁解牛看实现

2020-10-14 晁岳攀

Go 并发编程实战课 进入课程 >

讲述：安晓辉
时长 28:58 大小 26.53M

你好，我是鸟窝。

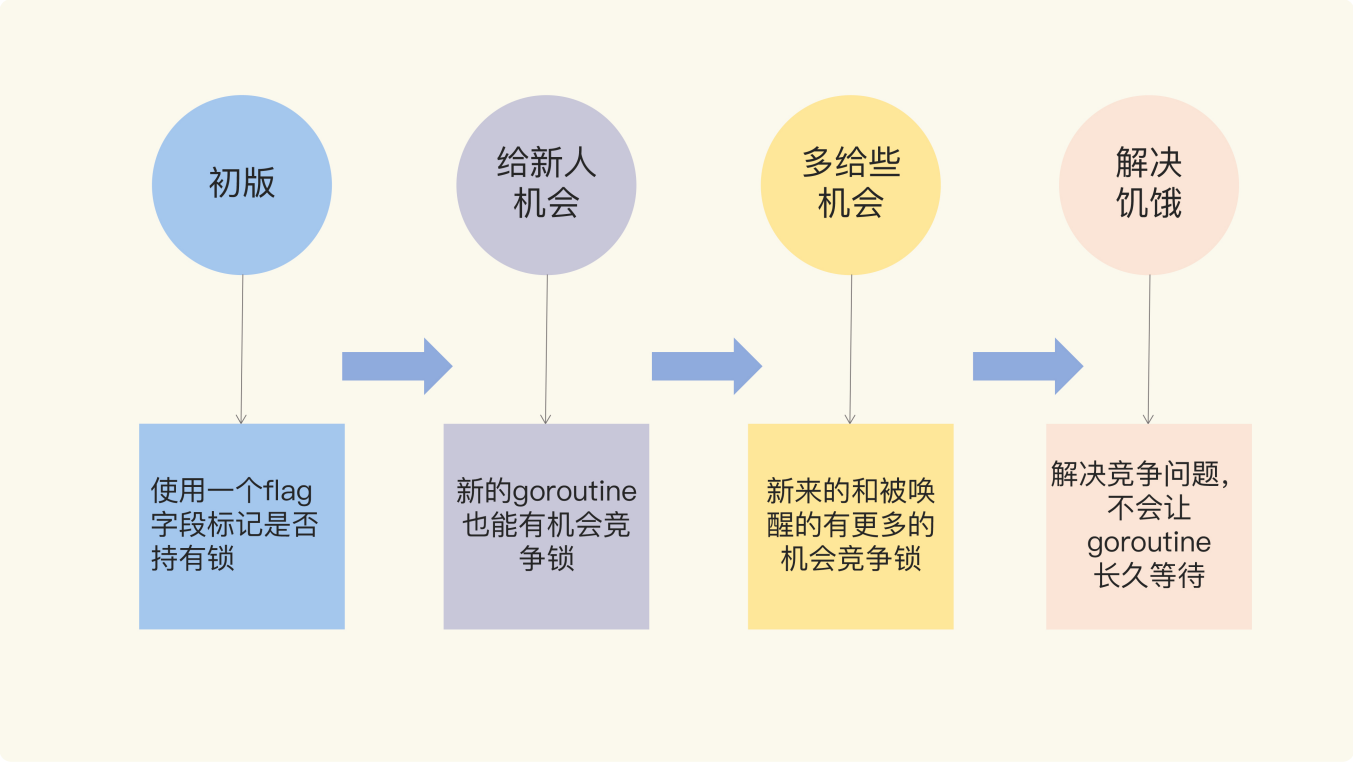
上一讲我们一起体验了 Mutex 的使用，竟是那么简单，只有简简单单两个方法，Lock 和 Unlock，进入临界区之前调用 Lock 方法，退出临界区的时候调用 Unlock 方法。这个时候，你一定会有一丝好奇：“它的实现是不是也很简单呢？”

其实不是的。如果你阅读 Go 标准库里 Mutex 的源代码，并且追溯 Mutex 的演进历史，你会发现，从一个简单易于理解的互斥锁的实现，到一个非常复杂的数据结构，这是一个逐步完善的过程。Go 开发者们做了种种努力，精心设计。我自己每次看，都会被这种[illegible]和精益求精的精神打动。

所以，今天我就想带着你一起去探索 Mutex 的实现及演进之路，希望你能和我一样体验到这种技术追求的美妙。我们从 Mutex 的一个简单实现开始，看看它是怎样逐步提升性能和公平性的。在这个过程中，我们可以学习如何逐步设计一个完善的同步原语，并能对复杂度、性能、结构设计的权衡考量有新的认识。经过这样一个学习，我们不仅能通透掌握 Mutex，更好地使用这个工具，同时，对我们自己设计并发数据接口也非常有帮助。

那具体怎么来讲呢？我把 Mutex 的架构演进分成了四个阶段，下面给你画了一张图来说明。

“**初版**”的 Mutex 使用一个 flag 来表示锁是否被持有，实现比较简单；后来照顾到新来的 goroutine，所以会让新的 goroutine 也尽可能地先获取到锁，这是第二个阶段，我把它叫作“**给新人机会**”；那么，接下来就是第三阶段“**多给些机会**”，照顾新来的和被唤醒的 goroutine；但是这样会带来饥饿问题，所以目前又加入了饥饿的解决方案，也就是第四阶段“**解决饥饿**”。



有了这四个阶段，我们学习的路径就清晰了，那接下来我会从代码层面带你领略 Go 开发者这些大牛们是如何逐步解决这些问题的。

初版的互斥锁

我们先来看怎么实现一个最简单的互斥锁。在开始之前，你可以先想一想，如果是你，你会怎么设计呢？

你可能会想到，可以通过一个 flag 变量，标记当前的锁是否被某个 goroutine 持有。如果这个 flag 的值是 1，就代表锁已经被持有，那么，其它竞争的 goroutine 只能等待；如果这个 flag 的值是 0，就可以通过 CAS（compare-and-swap，或者 compare-and-set）将这个 flag 设置为 1，标识锁被当前的这个 goroutine 持有了。

实际上，Russ Cox 在 2008 年提交的第一版 Mutex 就是这样实现的。

复制代码

```
// CAS操作，当时还没有抽象出atomic包
func cas(val *int32, old, new int32) bool
func semacquire(*int32)
func semrelease(*int32)
// 互斥锁的结构，包含两个字段
type Mutex struct {
    key  int32 // 锁是否被持有的标识
    sema int32 // 信号量专用，用以阻塞/唤醒goroutine
}

// 保证成功在val上增加delta的值
func xadd(val *int32, delta int32) (new int32) {
    for {
        v := *val
        if cas(val, v, v+delta) {
            return v + delta
        }
    }
    panic("unreached")
}

// 请求锁
func (m *Mutex) Lock() {
    if xadd(&m.key, 1) == 1 { //标识加1，如果等于1，成功获取到锁
        return
    }
    semacquire(&m.sema) // 否则阻塞等待
}

func (m *Mutex) Unlock() {
    if xadd(&m.key, -1) == 0 { // 将标识减去1，如果等于0，则没有其它等待者
        return
    }
    semrelease(&m.sema) // 唤醒其它阻塞的goroutine
}
```

这里呢，我先简单补充介绍下刚刚提到的 CAS。

CAS 指令将**给定的值**和**一个内存地址中的值**进行比较，如果它们是同一个值，就使用新值替换内存地址中的值，这个操作是原子性的。那啥是原子性呢？如果你还不太理解这个概念，那么在这里只需要明确一点就行了，那就是**原子性保证这个指令总是基于最新的值进行计算，如果同时有其它线程已经修改了这个值，那么，CAS 会返回失败**。

CAS 是实现互斥锁和同步原语的基础，我们很有必要掌握它。

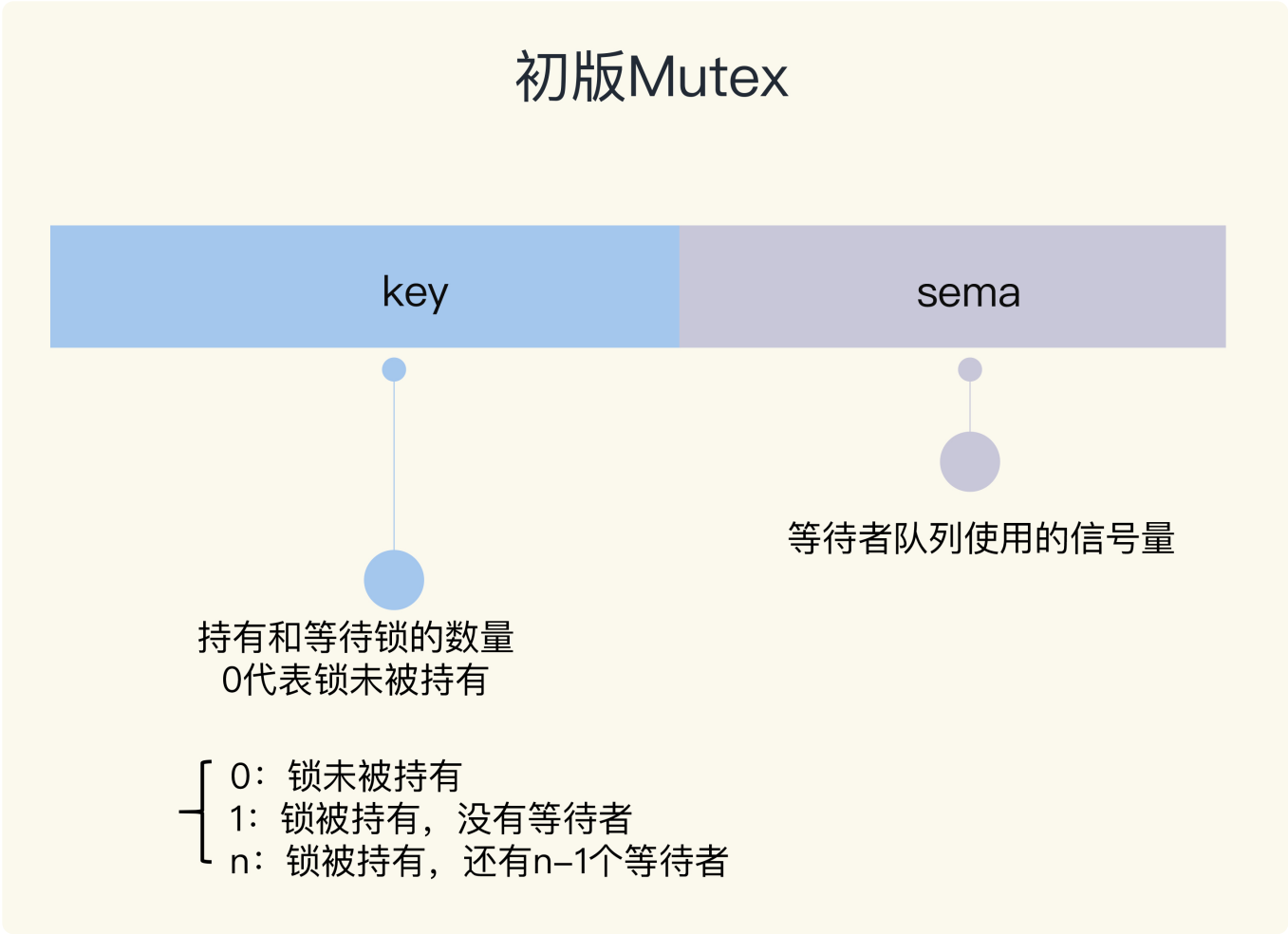
好了，我们继续来分析下刚才的这段代码。

虽然当时的 Go 语法和现在的稍微有些不同，而且标准库的布局、实现和现在的也有很大的差异，但是，这些差异不会影响我们对代码的理解，因为最核心的结构体（struct）和函数、方法的定义几乎是一样的。

Mutex 结构体包含两个字段：

字段 key：是一个 flag，用来标识这个排外锁是否被某个 goroutine 所持有，如果 key 大于等于 1，说明这个排外锁已经被持有；

字段 sema：是个信号量变量，用来控制等待 goroutine 的阻塞休眠和唤醒。



调用 Lock 请求锁的时候，通过 xadd 方法进行 CAS 操作（第 24 行），xadd 方法通过循环执行 CAS 操作直到成功，保证对 key 加 1 的操作成功完成。如果比较幸运，锁没有被别的 goroutine 持有，那么，Lock 方法成功地将 key 设置为 1，这个 goroutine 就持有了这个锁；如果锁已经被别的 goroutine 持有了，那么，当前的 goroutine 会把 key 加 1，而且还会调用 semacquire 方法（第 27 行），使用信号量将自己休眠，等锁释放的时候，信号量会将它唤醒。

持有锁的 goroutine 调用 Unlock 释放锁时，它会将 key 减 1（第 31 行）。如果当前没有其它等待这个锁的 goroutine，这个方法就返回了。但是，如果还有等待此锁的其它 goroutine，那么，它会调用 semrelease 方法（第 34 行），利用信号量唤醒等待锁的其它 goroutine 中的一个。

所以，到这里，我们就知道了，初版的 Mutex 利用 CAS 原子操作，对 key 这个标志量进行设置。key 不仅仅标识了锁是否被 goroutine 所持有，还记录了当前持有和等待获取锁的 goroutine 的数量。

Mutex 的整体设计非常简洁，学习起来一点也没有障碍。但是，注意，我要划重点了。

Unlock 方法可以被任意的 goroutine 调用释放锁，即使是没持有这个互斥锁的 goroutine，也可以进行这个操作。这是因为，Mutex 本身并没有包含持有这把锁的 goroutine 的信息，所以，Unlock 也不会对此进行检查。Mutex 的这个设计一直保持至今。

这就带来了一个有趣而危险的功能。为什么这么说呢？

你看，其它 goroutine 可以强制释放锁，这是一个非常危险的操作，因为在临界区的 goroutine 可能不知道锁已经被释放了，还会继续执行临界区的业务操作，这可能会带来意想不到的结果，因为这个 goroutine 还以为自己持有锁呢，有可能导致 data race 问题。

所以，我们在使用 Mutex 的时候，必须要保证 goroutine 尽可能不去释放自己未持有的锁，一定要遵循“**谁申请，谁释放**”的原则。在真实的实践中，我们使用互斥锁的时候，很少在一个方法中单独申请锁，而在另外一个方法中单独释放锁，一般都会在同一个方法中获取锁和释放锁。

如果你接触过其它语言（比如 Java 语言）的互斥锁的实现，就会发现这一点和其它语言的互斥锁不同，所以，如果是从其它语言转到 Go 语言开发的同学，一定要注意。

以前，我们经常会基于性能的考虑，及时释放掉锁，所以在一些 if-else 分支中加上释放锁的代码，代码看起来很臃肿。而且，在重构的时候，也很容易因为误删或者是漏掉而出现死锁的现象。

复制代码

```
type Foo struct {
    mu    sync.Mutex
    count int
}

func (f *Foo) Bar() {
    f.mu.Lock()

    if f.count < 1000 {
        f.count += 3
        f.mu.Unlock() // 此处释放锁
        return
    }

```

```
15      f.count++
16      f.mu.Unlock() // 此处释放锁
17      return
18  }
```

从 1.14 版本起，Go 对 defer 做了优化，采用更有效的内联方式，取代之前的生成 defer 对象到 defer chain 中，defer 对耗时的影响微乎其微了，所以基本上修改成下面简洁的写法也没问题：

复制代码

```
1  func (f *Foo) Bar() {
2      f.mu.Lock()
3      defer f.mu.Unlock()
4
5
6      if f.count < 1000 {
7          f.count += 3
8          return
9      }
10
11
12     f.count++
13     return
14 }
```

这样做的好处就是 Lock/Unlock 总是成对紧凑出现，不会遗漏或者多调用，代码更少。

但是，如果临界区只是方法中的一部分，为了尽快释放锁，还是应该第一时间调用 Unlock，而不是一直等到方法返回时才释放。

初版的 Mutex 实现之后，Go 开发组又对 Mutex 做了一些微调，比如把字段类型变成了 uint32 类型；调用 Unlock 方法会做检查；使用 atomic 包的同步原语执行原子操作等等，这些小的改动，都不是核心功能，你简单知道就行了，我就不详细介绍了。

但是，初版的 Mutex 实现有一个问题：请求锁的 goroutine 会排队等待获取互斥锁。虽然这貌似很公平，但是从性能上来看，却不是最优的。因为如果我们能够把锁交给正在占用 CPU 时间片的 goroutine 的话，那就不需要做上下文的切换，在高并发的情况下，可能会有更好的性能。

接下来，我们就继续探索 Go 开发者是怎么解决这个问题的。

给新人机会

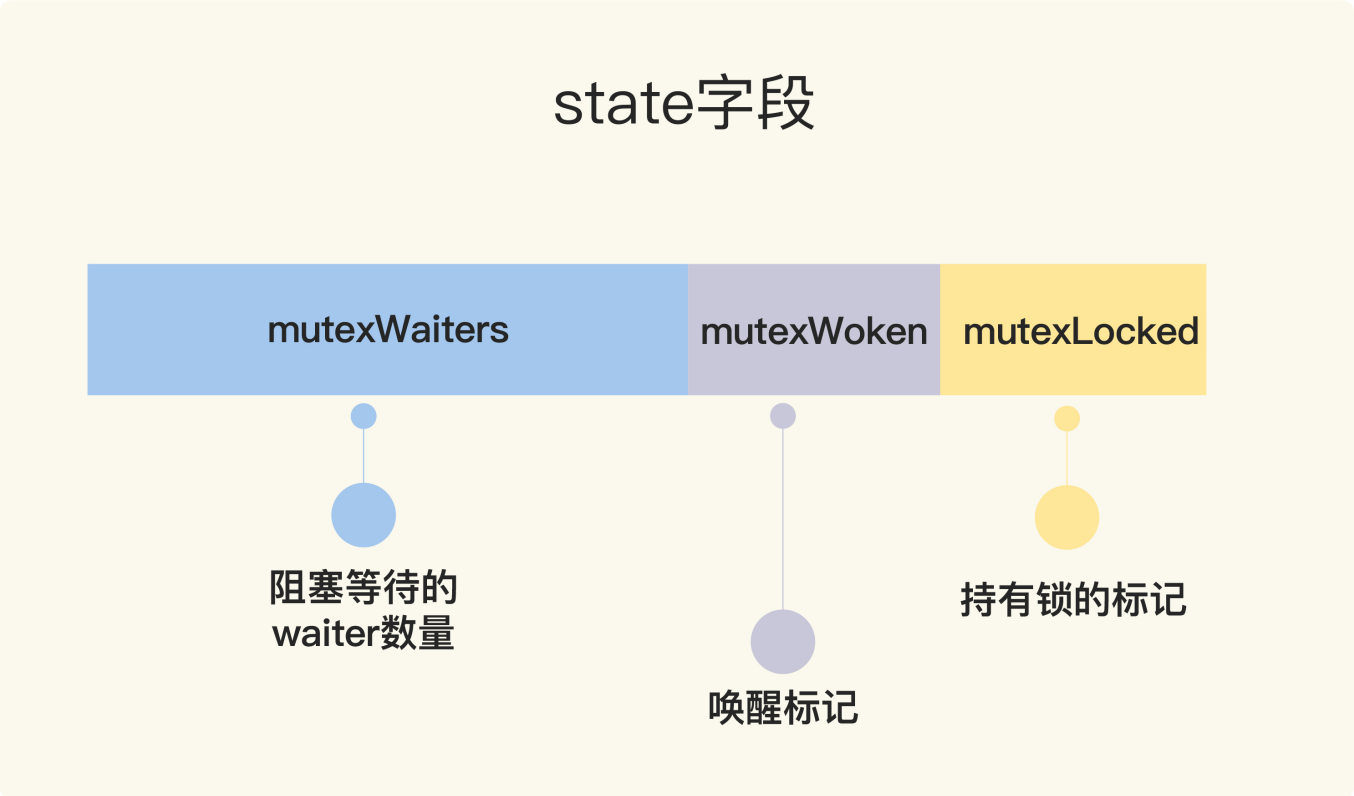
Go 开发者在 2011 年 6 月 30 日的 commit 中对 Mutex 做了一次大的调整，调整后的 Mutex 实现如下：

复制代码

```
type Mutex struct {
    state int32
    sema  uint32
}


const (
    mutexLocked = 1 << iota // mutex is locked
    mutexWoken
    mutexWaiterShift = iota
)
```

虽然 Mutex 结构体还是包含两个字段，但是第一个字段已经改成了 state，它的含义也不一样了。



state 是一个复合型的字段，一个字段包含多个意义，这样可以通过尽可能少的内存来实现互斥锁。这个字段的第一位（最小的一位）来表示这个锁是否被持有，第二位代表是否有唤醒的 goroutine，剩余的位数代表的是等待此锁的 goroutine 数。所以，state 这一个字段被分成了三部分，代表三个数据。

请求锁的方法 Lock 也变得复杂了。复杂之处不仅仅在于对字段 state 的操作难以理解，而且代码逻辑也变得相当复杂。

复制代码

```
1    func (m *Mutex) Lock() {
2         // Fast path: 幸运case，能够直接获取到锁
3         if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
4              return
5         }
6
7         awoke := false
8         for {
9              old := m.state
10             new := old | mutexLocked // 新状态加锁
11             if old&mutexLocked != 0 {
12                  new = old + 1<<mutexWaiterShift //等待者数量加一
13             }
14             if awoke {
15                  // goroutine是被唤醒的，
16                  // 新状态清除唤醒标志
17                  new &^= mutexWoken
18             }
19             if atomic.CompareAndSwapInt32(&m.state, old, new) {//设置新状态
20                  if old&mutexLocked == 0 { // 锁原状态未加锁
21                       break
22                  }
23                  runtime.Semacquire(&m.sema) // 请求信号量
24                  awoke = true
25             }
26        }
27    }
```

首先是通过 CAS 检测 state 字段中的标志（第 3 行），如果没有 goroutine 持有锁，也没有等待持有锁的 gorutine，那么，当前的 goroutine 就很幸运，可以直接获得锁，这也是注释中的 Fast path 的意思。

如果不够幸运，state 不是零值，那么就通过一个循环进行检查。接下来的第 7 行到第 26 行这段代码虽然只有几行，但是理解起来却要费一番功夫，因为涉及到对 state 不同标志位的操作。这里的位操作以及操作后的结果和数值比较，并没有明确的解释，有时候你需要根据后续的处理进行推断。所以说，如果你充分理解了这段代码，那么对最新版的 Mutex 也会比较容易掌握了，因为你已经清楚了这些位操作的含义。

我们先前知道，如果想要获取锁的 goroutine 没有机会获取到锁，就会进行休眠，但是在锁释放唤醒之后，它并不能像先前一样直接获取到锁，还是要和正在请求锁的 goroutine 进行竞争。这会给后来请求锁的 goroutine 一个机会，也让 CPU 中正在执行的 goroutine 有更多的机会获取到锁，在一定程度上提高了程序的性能。

for 循环是不断尝试获取锁，如果获取不到，就通过 runtime.Semacquire(&m.sema) 休眠，休眠醒来之后 awoke 置为 true，尝试争抢锁。

代码中的第 10 行将当前的 flag 设置为加锁状态，如果能成功地通过 CAS 把这个新值赋予 state（第 19 行和第 20 行），就代表抢夺锁的操作成功了。

不过，需要注意的是，如果成功地设置了 state 的值，但是之前的 state 是有锁的状态，那么，state 只是清除 mutexWoken 标志或者增加一个 waiter 而已。

请求锁的 goroutine 有两类，一类是新来请求锁的 goroutine，另一类是被唤醒的等待请求锁的 goroutine。锁的状态也有两种：加锁和未加锁。我用一张表格，来说明一下 goroutine 不同来源不同状态下的处理逻辑。

请求锁的goroutine	当前锁被持有	当前锁未被持有
新来的 goroutine	waiter++；休眠	获取到锁
被唤醒的 goroutine	清除mutexWoken标志； 重新休眠，加入等待队列	清除mutexWoken标志； 获取到锁

刚刚说的都是获取锁，接下来，我们再来看看释放锁。释放锁的 Unlock 方法也有些复杂，我们来看一下。

复制代码

```
1    func (m *Mutex) Unlock() {
2         // Fast path: drop lock bit.
3         new := atomic.AddInt32(&m.state, -mutexLocked) //去掉锁标志
4         if (new+mutexLocked)&mutexLocked == 0 { //本来就没有加锁
5             panic("sync: unlock of unlocked mutex")
6         }
7
8         old := new
9         for {
10            if old>>mutexWaiterShift == 0 || old&(mutexLocked|mutexWoken) != 0
11                return
12            }
13            new = (old - 1<<mutexWaiterShift) | mutexWoken // 新状态，准备唤醒gor
14            if atomic.CompareAndSwapInt32(&m.state, old, new) {
15                runtime.Semrelease(&m.sema)
16                return
17            }
18            old = m.state
19        }
20    }
```

下面我来给你解释一下这个方法。

第 3 行是尝试将持有锁的标识设置为未加锁的状态，这是通过减 1 而不是将标志位置零的方式实现。第 4 到 6 行还会检测原来锁的状态是否已经未加锁的状态，如果是 Unlock 一个未加锁的 Mutex 会直接 panic。

不过，即使将加锁置为未加锁的状态，这个方法也不能直接返回，还需要一些额外的操作，因为还可能有一些等待这个锁的 goroutine（有时候我也把它们称之为 waiter）需要通过信号量的方式唤醒它们中的一个。所以接下来的逻辑有两种情况。

第一种情况，如果没有其它的 waiter，说明对这个锁的竞争的 goroutine 只有一个，那就可以直接返回了；如果这个时候有唤醒的 goroutine，或者是又被别人加了锁，那么，无需我们操劳，其它 goroutine 自己干得都很好，当前的这个 goroutine 就可以放心返回了。

第二种情况，如果有等待者，并且没有唤醒的 waiter，那就需要唤醒一个等待的 waiter。在唤醒之前，需要将 waiter 数量减 1，并且将 mutexWoken 标志设置上，这样，Unlock 就可以返回了。

通过这样复杂的检查、判断和设置，我们就可以安全地将一把互斥锁释放了。

相对于初版的设计，这次的改动主要就是，新来的 goroutine 也有机会先获取到锁，甚至一个 goroutine 可能连续获取到锁，打破了先来先得的逻辑。但是，代码复杂度也显而易见。

虽然这一版的 Mutex 已经给新来请求锁的 goroutine 一些机会，让它参与竞争，没有空闲的锁或者竞争失败才加入到等待队列中。但是其实还可以进一步优化。我们接着往下看。

多给些机会

在 2015 年 2 月的改动中，如果新来的 goroutine 或者是被唤醒的 goroutine 首次获取不到锁，它们就会通过自旋（spin，通过循环不断尝试，spin 的逻辑是在runtime 实现的）的方式，尝试检查锁是否被释放。在尝试一定的自旋次数后，再执行原来的逻辑。

复制代码

```
1   func (m *Mutex) Lock() {
2       // Fast path: 幸运之路，正好获取到锁
3       if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
4           return
5       }
6
7       awoke := false
8       iter := 0
9       for { // 不管是新来的请求锁的goroutine，还是被唤醒的goroutine，都不断尝试请求锁
10          old := m.state // 先保存当前锁的状态
11          new := old | mutexLocked // 新状态设置加锁标志
12          if old&mutexLocked != 0 { // 锁还没被释放
13              if runtime_canSpin(iter) { // 还可以自旋
14                  if !awoke && old&mutexWoken == 0 && old>>mutexWaiterShift
15                      atomic.CompareAndSwapInt32(&m.state, old, old|mutexWok
16                      awoke = true
17                  }
18                  runtime_doSpin()
19                  iter++
20                  continue // 自旋，再次尝试请求锁
21              }
22              new = old + 1<<mutexWaiterShift
23          }
24          if awoke { // 唤醒状态
25              if new&mutexWoken == 0 {
26                  panic("sync: inconsistent mutex state")
27              }
```

```
28                  new &^= mutexWoken // 新状态清除唤醒标记
29              }
30              if atomic.CompareAndSwapInt32(&m.state, old, new) {
31                  if old&mutexLocked == 0 { // 旧状态锁已释放，新状态成功持有了锁，直接
32                      break
33                  }
34                  runtime_Semacquire(&m.sema) // 阻塞等待
35                  awoke = true // 被唤醒
36                  iter = 0
37              }
38          }
39      }
```

这次的优化，增加了第 13 行到 21 行、第 25 行到第 27 行以及第 36 行。我来解释一下主要的逻辑，也就是第 13 行到 21 行。

如果可以 spin 的话，第 9 行的 for 循环会重新检查锁是否释放。对于临界区代码执行非常短的场景来说，这是一个非常好的优化。因为临界区的代码耗时很短，锁很快就能释放，而抢夺锁的 goroutine 不用通过休眠唤醒方式等待调度，直接 spin 几次，可能就获得了锁。

解决饥饿

经过几次优化，Mutex 的代码越来越复杂，应对高并发争抢锁的场景也更加公平。但是你有没有想过，因为新来的 goroutine 也参与竞争，有可能每次都会被新来的 goroutine 抢到获取锁的机会，在极端情况下，等待中的 goroutine 可能会一直获取不到锁，这就是**饥饿问题**。

说到这儿，我突然想到了最近看到的一种叫做鹳的鸟。如果鹳妈妈寻找食物很艰难，找到的食物只够一个幼鸟吃的，鹳妈妈就会把食物给最强壮的一只，这样一来，饥饿弱小的幼鸟总是得不到食物吃，最后就会被啄出巢去。

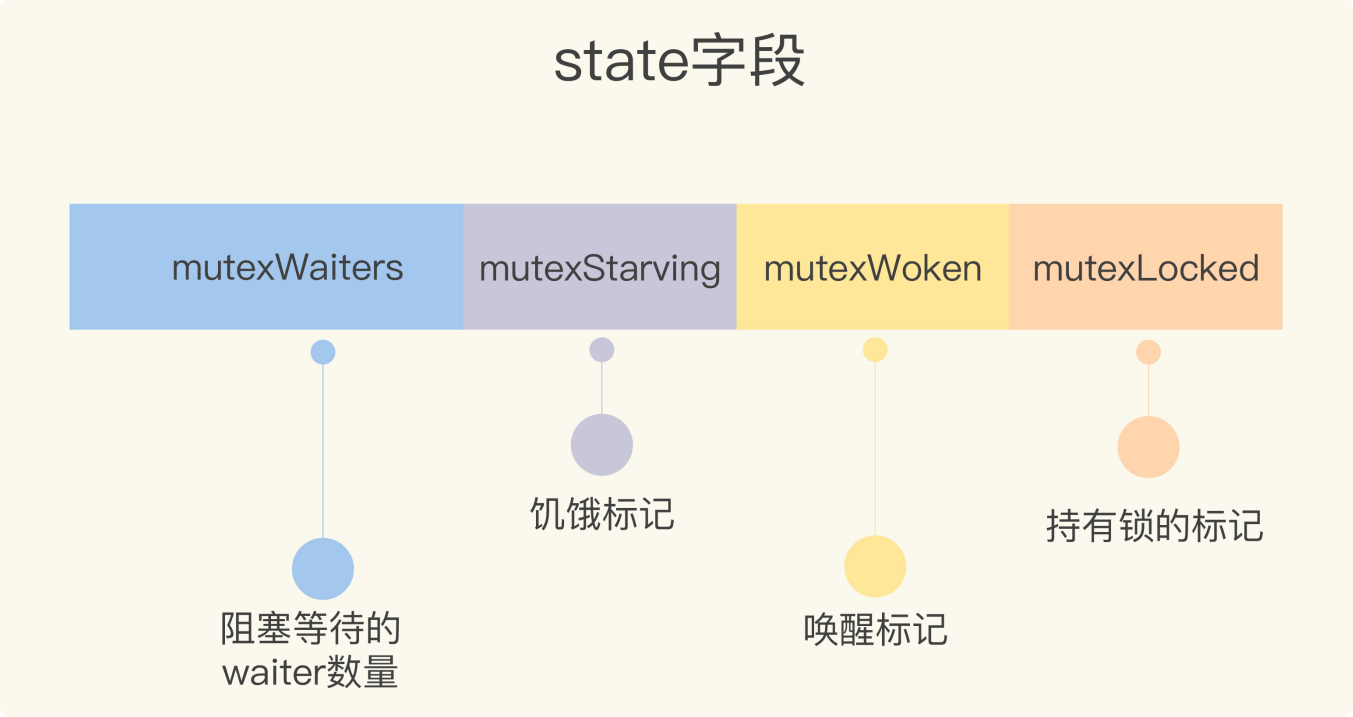
先前版本的 Mutex 遇到的也是同样的困境，“悲惨”的 goroutine 总是得不到锁。

Mutex 不能容忍这种事情发生。所以，2016 年 Go 1.9 中 Mutex 增加了饥饿模式，让锁变得更公平，不公平的等待时间限制在 1 毫秒，并且修复了一个大 Bug：总是把唤醒的 goroutine 放在等待队列的尾部，会导致更加不公平的等待时间。

之后，2018 年，Go 开发者将 fast path 和 slow path 拆成独立的方法，以便内联，提高性能。2019 年也有一个 Mutex 的优化，虽然没有对 Mutex 做修改，但是，对于 Mutex 唤醒后持有锁的那个 waiter，调度器可以有更高的优先级去执行，这已经是很细致的性能优化了。

为了避免代码过多，这里只列出当前的 Mutex 实现。想要理解当前的 Mutex，我们需要好好泡一杯茶，仔细地品一品了。

当然，现在的 Mutex 代码已经复杂得接近不可读的状态了，而且代码也非常长，删减后占了几乎三页纸。但是，作为第一个要详细介绍的同步原语，我还是希望能更清楚地剖析 Mutex 的实现，向你展示它的演化和为了一个貌似很小的 feature 不得不将代码变得非常复杂的原因。



当然，你也可以暂时略过这一段，以后慢慢品，**只需要记住，Mutex 绝不容忍一个 goroutine 被落下，永远没有机会获取锁。不抛弃不放弃是它的宗旨，而且它也尽可能地让等待较长的 goroutine 更有机会获取到锁。**

复制代码

```
type Mutex struct {
    state int32
    sema  uint32
}
```

```
6     const (
7         mutexLocked = 1 << iota // mutex is locked
8         mutexWoken
9         mutexStarving // 从state字段中分出一个饥饿标记
10        mutexWaiterShift = iota
11
12        starvationThresholdNs = 1e6
13    )
14
15    func (m *Mutex) Lock() {
16        // Fast path: 幸运之路，一下就获取到了锁
17        if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
18            return
19        }
20        // Slow path: 缓慢之路，尝试自旋竞争或饥饿状态下饥饿goroutine竞争
21        m.lockSlow()
22    }
23
24    func (m *Mutex) lockSlow() {
25        var waitStartTime int64
26        starving := false // 此goroutine的饥饿标记
27        awoke := false // 唤醒标记
28        iter := 0 // 自旋次数
29        old := m.state // 当前的锁的状态
30        for {
31            // 锁是非饥饿状态，锁还没被释放，尝试自旋
32            if old&(mutexLocked|mutexStarving) == mutexLocked && runtime_canSp
33                if !awoke && old&mutexWoken == 0 && old>>mutexWaiterShift != 0
34                    atomic.CompareAndSwapInt32(&m.state, old, old|mutexWoken)
35                    awoke = true
36                }
37                runtime_doSpin()
38                iter++
39                old = m.state // 再次获取锁的状态，之后会检查是否锁被释放了
40                continue
41            }
42            new := old
43            if old&mutexStarving == 0 {
44                new |= mutexLocked // 非饥饿状态，加锁
45            }
46            if old&(mutexLocked|mutexStarving) != 0 {
47                new += 1 << mutexWaiterShift // waiter数量加1
48            }
49            if starving && old&mutexLocked != 0 {
50                new |= mutexStarving // 设置饥饿状态
51            }
52            if awoke {
53                if new&mutexWoken == 0 {
54                    throw("sync: inconsistent mutex state")
55                }
56                new &^= mutexWoken // 新状态清除唤醒标记
57            }
```

```
58              // 成功设置新状态
59              if atomic.CompareAndSwapInt32(&m.state, old, new) {
60                  // 原来锁的状态已释放，并且不是饥饿状态，正常请求到了锁，返回
61                  if old&(mutexLocked|mutexStarving) == 0 {
62                      break // locked the mutex with CAS
63                  }
64                  // 处理饥饿状态
65
66                  // 如果以前就在队列里面，加入到队列头
67                  queueLifo := waitStartTime != 0
68                  if waitStartTime == 0 {
69                      waitStartTime = runtime_nanotime()
70                  }
71                  // 阻塞等待
72                  runtime_SemacquireMutex(&m.sema, queueLifo, 1)
73                  // 唤醒之后检查锁是否应该处于饥饿状态
74                  starving = starving || runtime_nanotime()-waitStartTime > star
75                  old = m.state
76                  // 如果锁已经处于饥饿状态，直接抢到锁，返回
77                  if old&mutexStarving != 0 {
78                      if old&(mutexLocked|mutexWoken) != 0 || old>>mutexWaiterSh
79                          throw("sync: inconsistent mutex state")
80                      }
81                      // 有点绕，加锁并且将waiter数减1
82                      delta := int32(mutexLocked - 1<<mutexWaiterShift)
83                      if !starving || old>>mutexWaiterShift == 1 {
84                          delta -= mutexStarving // 最后一个waiter或者已经不饥饿了，
85                      }
86                      atomic.AddInt32(&m.state, delta)
87                      break
88                  }
89                  awoke = true
90                  iter = 0
91              } else {
92                  old = m.state
93              }
94          }
95      }
96
97      func (m *Mutex) Unlock() {
98          // Fast path: drop lock bit.
99          new := atomic.AddInt32(&m.state, -mutexLocked)
100         if new != 0 {
101             m.unlockSlow(new)
102         }
103     }
104
105     func (m *Mutex) unlockSlow(new int32) {
106         if (new+mutexLocked)&mutexLocked == 0 {
107             throw("sync: unlock of unlocked mutex")
108         }
109         if new&mutexStarving == 0 {
```

```
110             old := new
111             for {
112                 if old>>mutexWaiterShift == 0 || old&(mutexLocked|mutexWoken|m
113                     return
114                 }
115                 new = (old - 1<<mutexWaiterShift) | mutexWoken
116                 if atomic.CompareAndSwapInt32(&m.state, old, new) {
117                     runtime_Semrelease(&m.sema, false, 1)
118                     return
119                 }
120                 old = m.state
121             }
122         } else {
123             runtime_Semrelease(&m.sema, true, 1)
124         }
125
```

跟之前的实现相比，当前的 Mutex 最重要的变化，就是增加饥饿模式。第 12 行将饥饿模式的最大等待时间阈值设置成了 1 毫秒，这就意味着，一旦等待者等待的时间超过了这个阈值，Mutex 的处理就有可能进入饥饿模式，优先让等待者先获取到锁，新来的同学主动谦让一下，给老同志一些机会。

通过加入饥饿模式，可以避免把机会全都留给新来的 goroutine，保证了请求锁的 goroutine 获取锁的公平性，对于我们使用锁的业务代码来说，不会有业务一直等待锁不被处理。

那么，接下来的部分就是选学内容了。如果你还有精力，并且对饥饿模式很感兴趣，那就跟着我一起继续来挑战吧。如果你现在理解起来觉得有困难，也没关系，后面可以随时回来复习。

饥饿模式和正常模式

Mutex 可能处于两种操作模式下：**正常模式**和**饥饿模式**。

接下来我们分析一下 Mutex 对饥饿模式和正常模式的处理。

请求锁时调用的 Lock 方法中一开始是 fast path，这是一个幸运的场景，当前的 goroutine 幸运地获得了锁，没有竞争，直接返回，否则就进入了 lockSlow 方法。这样的设计，方便编译器对 Lock 方法进行内联，你也可以在程序开发中应用这个技巧。

正常模式下，waiter 都是进入先入先出队列，被唤醒的 waiter 并不会直接持有锁，而是要和新来的 goroutine 进行竞争。新来的 goroutine 有先天的优势，它们正在 CPU 中运行，可能它们的数量还不少，所以，在高并发情况下，被唤醒的 waiter 可能比较悲剧地获取不到锁，这时，它会被插入到队列的前面。如果 waiter 获取不到锁的时间超过阈值 1 毫秒，那么，这个 Mutex 就进入到了饥饿模式。

在饥饿模式下，Mutex 的拥有者将直接把锁交给队列最前面的 waiter。新来的 goroutine 不会尝试获取锁，即使看起来锁没有被持有，它也不会去抢，也不会 spin，它会乖乖地加入到等待队列的尾部。

如果拥有 Mutex 的 waiter 发现下面两种情况的其中之一，它就会把这个 Mutex 转换成正常模式:

此 waiter 已经是队列中的最后一个 waiter 了，没有其它的等待锁的 goroutine 了；

此 waiter 的等待时间小于 1 毫秒。

正常模式拥有更好的性能，因为即使有等待抢锁的 waiter，goroutine 也可以连续多次获取到锁。

饥饿模式是对公平性和性能的一种平衡，它避免了某些 goroutine 长时间的等待锁。在饥饿模式下，优先对待的是那些一直在等待的 waiter。

接下来，**我们逐步分析下 Mutex 代码的关键行，彻底搞清楚饥饿模式的细节**。

我们从请求锁（lockSlow）的逻辑看起。

第 9 行对 state 字段又分出了一位，用来标记锁是否处于饥饿状态。现在一个 state 的字段被划分成了阻塞等待的 waiter 数量、饥饿标记、唤醒标记和持有锁的标记四个部分。

第 25 行记录此 goroutine 请求锁的初始时间，第 26 行标记是否处于饥饿状态，第 27 行标记是否是唤醒的，第 28 行记录 spin 的次数。

第 31 行到第 40 行和以前的逻辑类似，只不过加了一个不能是饥饿状态的逻辑。它会对正常状态抢夺锁的 goroutine 尝试 spin，和以前的目的一样，就是在临界区耗时很短的情况

下提高性能。

第 42 行到第 44 行，非饥饿状态下抢锁。怎么抢？就是要把 state 的锁的那一位，置为加锁状态，后续 CAS 如果成功就可能获取到了锁。

第 46 行到第 48 行，如果锁已经被持有或者锁处于饥饿状态，我们最好的归宿就是等待，所以 waiter 的数量加 1。

第 49 行到第 51 行，如果此 goroutine 已经处在饥饿状态，并且锁还被持有，那么，我们需要把此 Mutex 设置为饥饿状态。

第 52 行到第 57 行，是清除 mutexWoken 标记，因为不管是获得了锁还是进入休眠，我们都需要清除 mutexWoken 标记。

第 59 行就是尝试使用 CAS 设置 state。如果成功，第 61 行到第 63 行是检查原来的锁的状态是未加锁状态，并且也不是饥饿状态的话就成功获取了锁，返回。

第 67 行判断是否第一次加入到 waiter 队列。到这里，你应该就能明白第 25 行为什么不对 waitStartTime 进行初始化了，我们需要利用它在这里进行条件判断。

第 72 行将此 waiter 加入到队列，如果是首次，加入到队尾，先进先出。如果不是首次，那么加入到队首，这样等待最久的 goroutine 优先能够获取到锁。此 goroutine 会进行休眠。

第 74 行判断此 goroutine 是否处于饥饿状态。注意，执行这一句的时候，它已经被唤醒了。

第 77 行到第 88 行是对锁处于饥饿状态下的一些处理。

第 82 行设置一个标志，这个标志稍后会用来加锁，而且还会将 waiter 数减 1。

第 84 行，设置标志，在没有其它的 waiter 或者此 goroutine 等待还没超过 1 毫秒，则会将 Mutex 转为正常状态。

第 86 行则是将这个标识应用到 state 字段上。

释放锁（Unlock）时调用的 Unlock 的 fast path 不用多少，所以我们主要看 unlockSlow 方法就行。

如果 Mutex 处于饥饿状态，第 123 行直接唤醒等待队列中的 waiter。

如果 Mutex 处于正常状态，如果没有 waiter，或者已经有在处理的情况了，那么释放就好，不做额外的处理（第 112 行到第 114 行）。

否则，waiter 数减 1，mutexWoken 标志设置上，通过 CAS 更新 state 的值（第 115 行到第 119 行）。

总结

“罗马不是一天建成的”，Mutex 的设计也是从简单设计到复杂处理逐渐演变的。初版的 Mutex 设计非常简洁，充分展示了 Go 创始者的简单、简洁的设计哲学。但是，随着大家的使用，逐渐暴露出一些缺陷，为了弥补这些缺陷，Mutex 不得不越来越复杂。

有一点值得我们学习的，同时也体现了 Go 创始者的哲学，就是他们强调 GO 语言和标准库的稳定性，新版本要向下兼容，用新的版本总能编译老的代码。Go 语言从出生到现在已经 10 多年了，这个 Mutex 对外的接口却没有变化，依然向下兼容，即使现在 Go 出了两个版本，每个版本也会向下兼容，保持 Go 语言的稳定性，你也能领悟他们软件开发和设计的思想。

还有一点，你也可以观察到，为了一个程序 20% 的特性，你可能需要添加 80% 的代码，这也是程序越来越复杂的原因。所以，最开始的时候，如果能够有一个清晰而且易于扩展的设计，未来增加新特性时，也会更加方便。

思考题

最后，给你留两个小问题：

1. 目前 Mutex 的 state 字段有几个意义，这几个意义分别是由哪些字段表示的？
2. 等待一个 Mutex 的 goroutine 数最大是多少？是否能满足现实的需求？

欢迎在留言区写下你的思考和答案，我们一起交流讨论。如果你觉得有所收获，也欢迎你把今天的内容分享给你的朋友或同事。

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 01 | Mutex：如何解决资源并发访问问题?

下一篇 03 | Mutex：4种易错场景大盘点

精选留言 (39)

写留言

Junes

2020-10-14

老师讲得太棒了，我自己看Mutex源码时，没有前因后果，知识不成体系。交个作业：

1. 目前 Mutex 的 state 字段有几个意义，这几个意义分别是由哪些字段表示的?
和第四个阶段的讲解基本一致：前三个bit分别为mutexLocked、mutexWoken、mutex Starving，剩余bit表示mutexWaiter...

展开

作者回复: 赞学习态度，赞作业

21

锋
2020-10-14

老师您好，有个疑问。
runtime_Semrelease 信号唤起的总队queue中的第一个吗?
争抢锁在非饥饿模式下，是不是只有队首的waiter和新的goroutine之间发生?
谢谢

展开

作者回复: 是的。你可以看runtime中的实现https://github.com/golang/go/blob/0a820007e70fdd038950f28254c6269cd9588c02/src/runtime/sema.go#L321

4

buckwheat
2020-10-14

这源码看的感觉好难啊，尤其是这种并发的源码，老师有什么好的建议吗?

作者回复: Mutex和channel实现代码实现非常复杂，第一遍看不懂没关系，第二遍看不懂也没关系.....,永远看不懂也不影响你使用它。你可以每次只尝试理解一个知识点。我的经验是多看几遍，每一个if分支都理解它的意思，在纸上画一画state的值的变化

4

Dragon Frog
2020-10-25

老师，你好我有个疑问。在第二版中的 mutexWoken 这个含义到底该怎么理解。没办法很好的理解这个字段的作用

展开

作者回复: 标记是否有“通过unlock唤醒”的waiter在竞争锁

1

愚者
2020-10-20

```
if atomic.CompareAndSwapInt32(&m.state, old, new) {
        runtime.Semrelease(&m.sema) // 唤醒
        return
}
...
```

展开

1

(□ω□`)
2020-10-16

打卡 课后题：1：4个：mutexLocked代表是否被锁上 mutexWoken代表唤醒标记 mutexStarving代表是否处于饥饿 mutexWaiterShift 代表有多少go程在等待这个锁
2：int32 2^(32-3)-1 大约这么多 满足得了需求，每个go程2k 2k*2^29-1=1073741822k ≈ 1048575M≈1024G 我不信哪的内存能打到1T

展开

1

小龙虾
2020-10-14

打卡

展开

1

Geek_7ef408
2020-10-14

底层实现的代码和业务代码果然有很大不同，理解起来也更加困难，光是上面那一堆位运算就已经够让人头疼的了，全程懵逼的看完了这一篇，还需要好好消化一下

展开

作者回复: 没错，多来几遍，每次都有收获

1

星亦辰

2020-10-14

这一讲有点儿烧脑了，需要仔细看看

展开

作者回复: 是的，这一讲需要慢慢品，以后还可以翻过来回味一下

1

橙子888

2020-10-14

熬夜学习01课时后发现02课时也出来了，先来抢个沙发。

作者回复: 你第三讲打卡了，加油

1

lacfo

2020-10-27

老师好，我有1个关于第2.版的给新人机会的mutex的问题。在unlock中，您说或者别的goroutine已经加上了锁，就可以直接释放。我的问题是，如果我还没有释放，其它人是怎么拿到锁的呢？谢谢老师

作者回复: 释放锁的goroutine不必是锁的持有者

写点啥呢

2020-10-26

请问老师，在“给新人机会”这版本mutex中的Unlock方法的第8行，为什么要用减掉mutextLocked标志位为old，而不是重新读取m.state呢？如果在这行之前还有其他goroutine创建并获得锁，感觉这里的状态判断就不对了。请老师指点一下

展开

作者回复: 这里假定还没有其它giroutine获取到锁。
如果真的有goroutine获取到锁，那么第14行不成功，这个时候才读取state(第18行)

1

柠檬鱼也是鱼
2020-10-24

感觉自己只能看看大概思路，有收获就行

展开

pony
2020-10-24

自己设计互斥锁只能想到第一版，跟着老师看了源代码发展过程，才发现提高并发效率要考虑的因素好多，公平性、高效性之间的追求无极限。并发中队头阻塞感觉现在的mutex也是存在的

myrfy
2020-10-23

老师，使用Mutex后就可以放心大胆的操作共享的结构体了吗？不是很清楚Go的Runtime在后面有没有其他的机制。

比如我有这样一个结构体

...

展开

作者回复: 建议看一下go内存模型。mutex有happen before保证

极夜
2020-10-22

就喜欢这样有历史厚度的文章。值得点赞，值得购买！

展开

大恒数据
2020-10-21

请问 new &^= mutexWoken 怎么运算呢？第一次见两个操作符在一起

作者回复: go规范中位操作

Money
2020-10-21

老师，问下，第一版的为什么不直接使用计数器sema作为互斥锁，初始值为1？

结构体如下
type Mutex struct {
    sema int32 // 信号量专用，用以阻塞/唤醒goroutine ...
展开

作者回复: sema主要内部用来阻塞等待的队列

Black
2020-10-21

runtime.go 这个文件里的函数，老师有推荐的资料学习一下不

作者回复: 网上都是一些关注某个点的介绍。你可以学到哪个点就搜哪个点

Geek_3533e2
2020-10-20

多给些机会中的Lock方法中，有一部分代码不是特别理解：
```
if old&mutexLocked != 0 { // 锁还没被释放
            if runtime_canSpin(iter) { // 还可以自旋
                if !awoke && old&mutexWoken == 0 && old>>mutexWaiterShift !...
```
展开

作者回复: 没有唤醒的goroutine和它抢的话，它把自己"伪装"成唤醒的goroutine